Hauptseminar Rechnernetze

An Overview Of Model Driven Architecture

Thomas Zimmermann

30th May 2008
Dresden University of Technology
Faculty of Computer Science
Institute for System Architecture
Chair for Computer Networks

Abstract

This document gives an overview of the state of Model Driven Architecture (MDA) as of spring 2008. It discusses the basic ideas of MDA, its theoretical concepts and relation to similar technologies. Some tools are evaluated and its critic's points are presented. The paper's intended audience are software developers who are new to MDA and want to have an introduction to the topic.

1 Introduction

Model-driven software development is a technique for creating software applications where a problem's solution is not explicitly stated in a classic programming language, but modeled with some high-level modeling language.

Model-driven software development is based on the concepts of modeling of software and the automatic generation of source code. This section presents each of these concepts by an example and then introduces Model Driven Architecture as an attempt to standardize model-driven software development.

1.1 An Example of Modeling

A *model* is an abstract, idealized representation of some concrete knowledge. Because most problems are too complex to be fully understood, models often focus on the major points of the topic, but leave out most minor details.

The obvious example of modeling in software development is the use of the Unified Modeling Language (UML). It allows the description of software designs from some high-level perspective.

UML is independent from any programming language. This enables developers to conceive the structure and behavior of an application, without the need to care about implementation details.

On the other hand, the fact that UML focuses on object-oriented programming is a good example of the limitations of models. An application can be designed along the paradigms of functional programming, but UML does not allow for a simple description of such designs.

1.2 An Example of Automatic Source-Code Generation

The process of automatic generation of source code follows some simple steps. The programmer

1. describes the problem in some very expressive high-level notation,
2. uses some tools to automatically generate source code from the description, and
3. integrates the results into the application.

An example of automatic source-code creation is the development of a text parser. The parser reads text from an input source, e.g. a file on the user's hard disk, and validates its syntactical correctness against some predefined rules, i.e. a grammar.

With the generative approach, the programmer does not implement the text parser directly, but describes the text file's grammar by some specialized domain-specific language.

The common language for representing context-free grammars is the *Extended Backus-Naur form* (EBNF) [Int96]. The translation to source code is implemented by a generic conversion tool, i.e. a parser generator. This program takes an EBNF grammar description as its input and outputs the parser's source code, i.e. a finite state automaton. The generated automaton is then included in the application's source code.

This example is very common and has been standard practice for several decades[1]. It perfectly outlines the advantages of automatic source-code creation.

- The specialization of the EBNF makes it very easy to describe and change grammars. This way, even non-programmers are able to write text file parsers.
- The EBNF is independent of any programming language or environment. All such platform-specific details are handled by the parser generator. This also allows for optimizing the automaton for some metric, e.g. runtime speed or memory consumption, depending on the underlying computer system.
- The parser generator has to be written only once. It can then be used with any grammar description. The error-prone details of the resulting automaton's implementation are handled by the generator.

Combining the properties of code generation with the modeling of software finally results in model-driven software development and Model Driven Architecture.

[1]A widely used parser generator for Unix systems is GNU Bison [GNU]. Its history reaches back to the 1970s.

1.3 Model Driven Architecture

The *Model Driven Architecture*[2] (MDA) [OMGa] is an attempt to standardize model-driven software development on a number of common technologies and file formats.

MDA is defined and maintained by the *Object Management Group* (OMG) [OMGb], a consortium formed by several hundred companies from the IT industry. The OMG also maintains other industry standards, notably the *Common Object Request Broker Architecture* (CORBA) [OMGd] and the *Unified Modeling Language* (UML) [OMGc]. The later is closely related to MDA.

MDA provides a generalization of modeling and code generating. The idea of Model Driven Architecture is to define highly-detailed models of applications and generate the source code automatically.

Modeling is done in UML or some other special-purpose language. The MDA standard explicitly allows for the definition of new modeling languages.

Model Driven Architecture is advertised with the following advantages.

Focus on models. In the planning stage of software development, the problem and its solution has to be modeled, either because the problem is to complicated to be manageable as a whole or just to make sure that everyone involved shares the same idea of the final application.

Programmers tend to produce software that differs from its model, because what looked good at the planning stage is sometimes not implementable in the concrete programming language.

Because some modeling has to be done anyway, the idea is to reuse these models to generate the actual implementation. Therefore, the model has to be specified in many details and a description of its runtime behavior has to be added.

If done consistently, many or even all components of the application can be generated by some transformation tool. Depending on the model's level of detail, the OMG claims to produce 50% to 100% automatically.

Concentrate on business logic. *Business logic* are those parts of an application that contain the knowledge that is specific to the problem domain.

The idea is that developers, using MDA, can concentrate on these core components and let the low-level routines and glue code be generated automatically. In the end, they are faster to produce software then their competitors and get their product to the market sooner.

Platform independence. Platform independence is achieved by having models that are completely abstract from any concrete technology platform. This makes it possible to deliver applications for several platforms without any overhead in maintenance. Additionally it is possible to migrate between platforms easily whenever an old platform becomes obsolete and a new one evolves.

1.4 Overview

This section introduced model-driven software development and some of its basic terms.

Section 2 gives an overview of some technologies which are related to model-driven software development.

[2]The OMG incorrectly calls this standard *Model Driven Architecture* instead of *Model-Driven Architecture*. This document follows the OMG's convention.

Section 3 describes the concepts of Model Driven Architecture and how they are intended to work together.

Some tools are evaluated in Section 4.

Model Driven Architecture is not without critics. Their main points are presented in Section 5.

A conclusion is given in Section 6.

A list of all external references is found at the end of this document.

2 Related technologies

The idea of creating software automatically is neither new nor original to model-driven software development. This section gives an overview of related technologies.

Compilers. A compiler's task is to transform a program from one language into a semantically equivalent program in another language. This process is called compiling.

Compilers are among the oldest tools in computer science. The first compiler was written in 1952 by Grace Hopper for the programming language A-0. In the 1960s, John Backus' FORTRAN became the first wide-spread programming language. Another notable step was the development of C by Dennis Richie in 1969. This is todays standard in system programming and influenced many later programming languages. The object-oriented style of programming became popular in the 1980s and 1990s with C++ by Bjarne Stroustrup and Java by James Gosling.

The common use of compilers is to transform statements from high-level, machine-independent programming languages into low-level, machine-dependent instructions. Code generators in model-driven software development do the same on a more abstract level. Their main difference to common compilers is that their input languages are not of general purpose, but highly specialized to certain problems.

More information about compilers is found in [ASUL06], a classic book on that topic.

CASE tools. *Computer-Aided Software Engineering* (CASE) is the idea of using software to develop and maintain other software. Such tools provide an environment to plan, model and describe software solutions to a given problem. One important aspect hereby is the graphical notation. The CASE tool creates, at least partially, the software that solves the presented problem.

CASE tools where invented at the beginning of the 1980s. They had there most popular time at the beginning of the 1990s, but mostly failed because they delivered to little to be useful for general software development.

Ideas of CASE tools are found in today's Integrated Development Environment, e.g. planning and modeling facilities. CASE tools are often seen as the ancestors of Model Driven Architecture.

XML. The *Extensible Markup Language* (XML) [W3C] is a file format for representing hierarchically-structured information.

The hierarchy is formed by tags. Each tag can hold a set of attributes, where each attribute can hold a value. The notation is independent from the represented information as it only defines the syntax, but leaves semantics up to the actual application. This gives XML a high flexibility and allows for the independence of XML parsers and stored data.

XML was invented by the *World Wide Web Consortium* (W3C) in 1998. It is the successor of the *Standard Generalized Markup Language* (SGML). XML was rapidly adopted as the standard syntax for file formats in many domains.

Though XML is not directly related to model-driven software development, it is of some importance. Part of the Model-Driven-Architecture specification is the *XML Metadata Interchange* (XMI), an XML-based file format for storing modeling information.

Despite from information storage, XML is frequently used for controlling application behavior. Program components are described in XML notation and then generated automatically by the application at runtime. This technique shows strong similarities to model-driven development.

An example is the creation of graphical user interfaces. Instead of explicitly building an interface with source code, its appearance is described with XML syntax. Each tag describes one element of the interface, e.g. a button, menu, or list. The application loads this description and builds the interface accordingly. An example for an XML interface builder is Glade [GLA].

Templates and Generics. Templates are part of C++ [Str00] where they allow for the parameterized implementation of classes and functions. Instead of explicitly implementing a class, the programmer implements a template of the class where some elements are given as parameters. e.g. data types of properties, static array bounds, or function objects.

To instantiate a template, the programmer supplies a concrete value for each parameter. The specialized template now defines a C++ class or function and can be used as such. This allows for a very high flexibility and reusability, while the source code is still type safe and fast.

An example of a template is the string implementation that comes with C++. The string is independent from the underlying data type of its characters. It can be used similar to a common C string when instantiated with *char*, but can also act as a wide string when instantiated with *short*.

The template mechanism is related to model-driven software development in such that the compiler acts as a code generator when a template gets instantiated. The template mechanism is Turing-complete and can be used for functional programming at compile time.

Generics are a mechanism of the Java programming language [GJSB05]. They allow for the lexical substitution of tokens at compile time. Generics do not have the expressive power of C++ templates, but are similar to preprocessors in other languages.

UML 1. Version 1 of UML [Int05] is the starting point from which Model Driven Architecture was developed. It was defined in the 1990s to unify several other notations for the modeling and design of software. Today, it is the standard notation in this area.

Unlike version 2, UML 1 has no capabilities for code generation and mainly focuses on modeling. It allows for the description of application states and some simple behavior, but is not expressive enough to model complete applications.

Many extensions from version 1 to 2 of UML were made to support Model Driven Architecture. Most of them are related to the description of application behavior. Additionally, the language was redefined as an instance of the Meta Object Facility.

3 Concepts

This section presents the basic concepts of Model Driven Architecture.

3.1 Meta Object Facility

The *Meta Object Facility* (MOF) is the standard that defines the concepts of modeling within the Model Driven Architecture. It describes a hierarchy of 4 layers which are upwardly named M0 to M3. The properties and behaviors of each layer are defined in the layer on top of it. Figure 1 illustrates this.

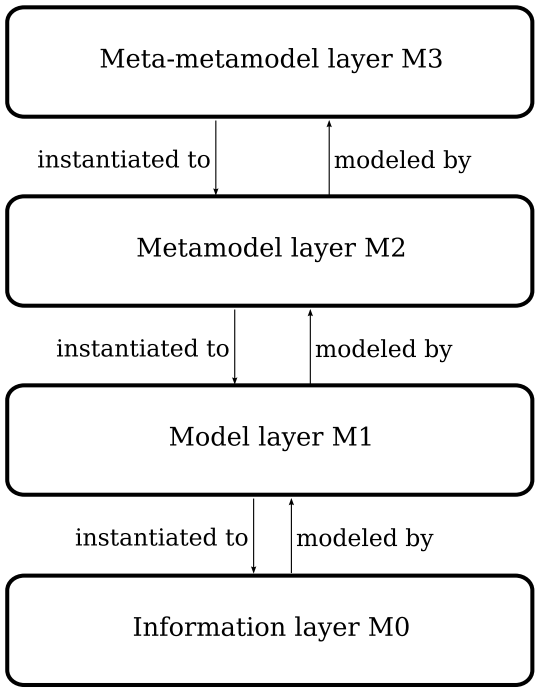


Figure 1: The hierarchy of modeling layers as defined by the Meta Object Facility. Each layer contains instances of the models in the layer above.

3.1.1 Information layer M0

The *information layer* M0 can be seen as *the program in execution*, i.e. an instance of the modeled application. The output of M0 is therefore the solution of the developer's original real-world problem.

A program's instance in M0 consist of

- an executable representation,
- the program's current state, and
- a set of resources, occupied by the program.

An example of an application's representative in M0 is a running Java Virtual Machine. It consists of a set of instructions, which have been loaded from an external source, a stack that holds the processes state, and its resources, e.g. virtual memory, open files, or network connections.

The Meta Object Facility does not specify the M0 layer in detail. This is an implementation detail of the MDA framework. One implementation could execute directly on the hardware, while another might use some virtual machine or high-level framework.

3.1.2 Model layer M1

The *model layer* M1's purpose is to allow for the description of real-world problem to the computer system. It is associated with instances of modeling languages, e.g. an instance of an UML class.

The Meta Object Facility does not dictate a modeling language. It encourages the use of several distinct languages where each language is optimized towards a specific problem domain. For object-oriented designs, the MOF already defines the Unified Modeling Language. Section 3.1.3 discusses this topic in more detail.

Because complete modeling of an application is often not possible, the layer M1 also has to be associated with programming languages, e.g. Assembler, C, C++, or Java.

In programming languages there exists data structures and algorithms. Data structures are used to represent entities of the real world in a way that is accessible by the computer. Algorithms connect data structures and describe the interactions between them.

In Model Driven Architecture, data structures and algorithms often do not exist explicitly, i.e. in form of a programming language. If possible, any aspect of an application should be contained in the model. A concrete implementation is generated by a transformation. Section 3.2 describes this process in more detail.

An instance of a program's M1 representation forms a model in M0. The instantiation occurs when user executes the program.

3.1.3 Metamodel layer M2

A *metamodel* is a model that describes another model.

The *metamodel layer* M2 contains the syntax and semantics of modeling languages. These are used for application modeling in layer M1.

The intention is to allow for the definition of *domain-specific languages* (DSL), i.e. very expressive notations which are optimized for certain problems. This makes the modeling in layer M1 very compact. For example, the EBNF, that was used in the introduction, is a DSL for the description of context-free grammars.

The MOF specifies version 2 of the Unified Modeling Language (UML) as part of this layer. UML allows for the description of a problem in a more abstract way then common programming languages do. The developer hereby specifies the program's components, e.g classes, and their interaction in UML notation. UML provides structural and behavioral constructs for doing this.

Structural constructs are used to define static properties and structures of the program. In layer M1, this is used to model classes of an object-oriented programming language, or data structures in general. UML provides notations to model different relations between classes or classes and their properties. Some examples are

- associations,
- generalizations,
- attributes of classes, or
- multiplicities of properties.

Version 2 of UML was extended to make the description of a model's semantics possible. Such *behavioral constructs* describe the behavior of the program and the active interaction between its structural constructs. This can be associated with the program's logic or algorithms. Such statements are generally referred to as *Executable UML*. It allows for the modeling of behavior such as

- actions,
- activities, i.e. sets of actions,
- state and state machines, or
- use cases.

UML also allows for the definition of *profiles*, a mechanism to extend its capabilities. Profiles are sufficient for the specialization or extension of existing constructs, but do not allow for the definition of completely new ones.

3.1.4 Meta-metamodel layer M3

The top-most layer is the *meta-metamodel layer* M3. It is used to define distinct domain-specific languages. The MOF specification is rather vague about M3 and does not even require its presence.

Defining constructs in M3 would in principal require a hypothetical layer M4 which would itself need a hypothetical layer M5. and so on. To prevent this kind of *infinite recursion*, the Meta Object Facility specifies the use of UML for the definition of new DSLs. This is possible, because UML itself is not only part of the MOF, but also described by its own external specification.

3.2 Models, Platforms and Transformations

An overview of the theoretical approach to modeling has been presented in the previous sub-section. Now the focus is laid on several aspects of this process.

3.2.1 Models

The programmer models the application as a *platform-independent model* (PIM). As the name suggests, this model does not expose details of an underlying platform.

Platform-independent modeling is done with the languages from the metamodel layer of the MOF stack. The OMG advertises the UML to describe complete object-oriented software designs, i.e. classes, class hierarchies, interaction between them, as well as the behavior of the system in certain situations.

The PIM cannot be executed directly. It is therefore transformed to a *platform-specific model* (PSM). Each PSM is specific to the platform that it was created for. It can only be executed on top of this platform.

3.2.2 Platforms

A *platform*, in the sense of MDA, is a base on where to generate platform-specific models. This does not have to be a hardware platform. Instead, it is more common to create PSMs on top of the Java Environment or some high-level domain-specific system.

Finding the right kind of abstraction for a platform can be complicated. If the platform is very generic, it is possible to model a wide range of applications on top of it. On the other hand, the transformation from the PIM to the PSM can become quite large, because many details have to be handled. The Java Environment is an example thereof.

The opposite are platforms that are very domain-specific. Modeling transformations on top of these is rather simple if the platform and the model relate to the same problem domain. Doing anything else is hard or even impossible because the platform simple lacks the necessary facilities.

Platforms can roughly be divided into three groups.

Components. Problem-oriented service libraries are commonly referred to as *components.* Components provide a set of functions that are imported into an application. The application calls them to access the provided service. An example of components are graphics-rendering libraries, such as OpenGL [Khr].

Frameworks. A framework is a highly-configurable library that implements a generic solution to a general problem. The application can register callback functions to adapt the framework to a special instance of the problem. The difference between components and frameworks is that components are called by the application frequently; frameworks are configured once and then execute automatically. During the execution the framework calls back into the application via the provided callback functions. An example of frameworks are GUI widget libraries.

Middleware. Some platforms are neither components nor frameworks. They are referred to with the more general term *middleware.*

3.2.3 Transformations

A *transformation* maps platform-independent models to platform-specific ones. This process is also called *model-to-model transformation.* It is done automatically by a *transformation tool.*

Transformations are part of the layer M1 of the MOF hierarchy. A transformation only changes the model's syntactical representation, but not its semantics. Hence, transformations do not move between layers. Other layers can contain transformations too, but these are not covered by the MOF standard. An example is some self-modifying code on layer M0.

Transformations generally work in the following way.

1. The transformation tool reads the platform-independent model which is written in a domain-specific language.
2. The tool checks for the input's correctness.
3. If it is correct, the transformation tool builds an internal representation.
4. For each element in this representation, there is a rule of how to map it onto the target platform in a semantics-preserving way. The transformation tool applies these transformation rules.
5. The generated platform-specific model is written.

This is similar to the functionality of a compiler[3].

The changes in syntax can be quite extensive. For example, in the introduction, an EBNF grammar description was transformed into an implementation of a finite state automaton.

It is possible that a platform-specific model is not instantly executable. More transformations can be applied or the model can be linked with hand-written code until the final application has been constructed. Again, all these actions do not move between layers.

Because it is generated automatically, the platform-specific model should not be edited. If it happens that the PSM needs to be changed by hand, a better approach is to extend the syntax and semantics of the domain-specific language to allow for the solution of the problem in the PIM.

4 Tools

The OMG's website lists several tools for aiding MDA-based software development. In this section a small evaluation of some of them is presented. It is indented to give the reader a general impression of what to expect when starting with Model Driven Architecture.

The test candidates are Eclipse, openArchitectureWare, and AndroMDA. Eclipse is an Integrated Development Environment

[3]The only reason to name this process *transformation* instead of *compiling* is to distinguish modeling from common programming.

and widely used by Java programmers. AndroMDA and openArchitectureWare promote themselves as leading tools for model-driven software development. Each of these tools is in a mature state and appears to be suitable for production use.

With each program some simple beginner's tasks are tried out. This consists of 1. installing the software, 2. using some provided tutorial to create an application, and 3. trying to extend the application in a model-driven way. All tests where executed on a Windows 2000 machine with the Sun's Java Development Kit 6.0 installed.

The author has been a Unix user for approximately 10 years. He has been writing software for University and personal use in C, C++ and Java in this environment. He is a complete beginner to each of the applications, and does not have had any practical experience with Model Driven Architecture before.

Please keep in mind that most of this section is based on subjective experience.

4.1 Eclipse

Eclipse [Ecla] is an Integrated Development Environment, available for Linux, Macintosh and Windows. Eclipse is available as open-source software under the terms of the *Eclipse Public License.*

Model-driven software development in Eclipse can be done with the *Eclipse Modeling Framework* (EMF). This is a set of plugins for Eclipse that include generators for models and code.

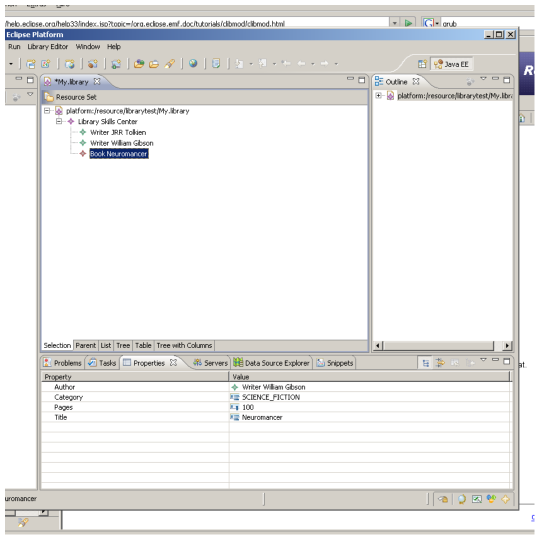
Installation. The installation procedure on Windows is very easy and consists of downloading and decompressing an archive file. Several pre-packaged editions of Eclipse are available for download. They are distinct from each other by the number of included extensions. The Enterprise Edition 3.3.2 was chosen, as it comes with all plugins and extensions available.

Basic usage. The second part of this test is based on a tutorial [Eclb] for beginners, available at the Eclipse website. It introduces the EMF and shows how to create a simple application for managing a library. Users can add books and writers and connect them. The application is executed as a plugin in Eclipse.

The tutorial's first step is to to create a *generator model*, i.e. the model that is later used to generate the application. Therefore, the programmer has to write a Java interface for book and writer classes. The interface is annotated with some extra information that allows the generator tool to fill in the code and connect both classes. For example, each book has a property *writer* and each writer has a property *books*, which is a list of associated books. The annotations allow for the automatic matching of both properties. If the user selects a specific author for a book, the book automatically appears in the writers list of books. The EMF framework generates all necessary code.

In the second step, the generator model is used to create a plugin for Eclipse that provides the application's functionality to the user. Again, the EMF framework creates the source code automatically. The resulting application is shown in figure 2.

Extending. The third part of the test is about adding a new component in a model-driven fashion. The author decided to add a new class *Publisher* that holds the name of the publisher and a list of published books. Additionally, each book should become a new property *publisher*.



```
platform:/resource/librarytest/My.library
  Library Skills Center
    Writer JRR Tolkien
    Writer William Gibson
    Book Neuromancer
```

Property	Value
Author	Writer William Gibson
Category	SCIENCE_FICTION
Pages	100
Title	Neuromancer

Figure 2: The generated plugin runs in Eclipse. The upper area shows the books and writers in the library, the lower area allows for the editing of their properties.

First it was tried to extend the generator model which appeared to be the most model-driven style. Unfortunately, the author has not been able to figure out how to do this. The generator model seems to be static and not editable.

Then it was tried to extend the annotated Java interfaces. The original interfaces where partially lost in the process of creating the generator model. Therefore, changes were added to the automatically created versions. Rebuilding the plugin as described in the tutorial worked seamlessly. However, it was not possible to execute the new version. Eclipse only provided the old plugin, which had no information about publishers.

Conclusion. On the positive side is that following the beginner's tutorial is really easy. Impressive were the ease of installation and the quality of the tools which worked out-of-the-box and exactly as described in the text. No obvious bugs or errors in the process were discoverable.

On the negative side is that the annotated Java interfaces got overwritten when the generator model were created. This destroyed the original interface code. The author does not think that destroying handwritten code is an acceptable behavior. The second issue is that extending the model was a failure. Maybe this is possible, but it was not obvious of how to do it. The fact that the original Java interfaces were lost made it even impossible to start from scratch.

Readers who already use Eclipse should not have much work to try out EMF and see if it fits their needs. At least it could be useful for small or minor tasks, e.g. generating glue code between components.

4.2 AndroMDA

AndroMDA [Andb] is an MDA generator framework. It is advertised with features of

- modularity,
- support for various UML tools,
- support for UML 1.4, and
- support for many different platforms.

AndroMDA is licensed under the terms of the *Berkeley Software Distribution* license.

Installation. The installation and usage is described by a tutorial [Anda] on AndroMDA's website. It is very comprehensive and describes every step in detail.

A possible problem is, that there are different versions of the tutorial. The author first tried a version that seemed to be up-to-date, but actually was too old to be useful anymore. So the test had to be restarted with a later version.

The installation of AndroMDA 3.2 is extensive and includes installing a package manager, a database system, an UML modeling tool, and several other, minor components. It is notable that there are alternative implementations for most of these applications, including some with source code available. It is unlikely that users will ever encounter a vendor-lock-in scenario where they depend on the availability of one concrete tool.

Basic usage. The tutorial describes the creation of an application for managing time tables. It is modeled with an UML modeler and transformed into platform-specific source code by AndroMDA.

Again, the tutorial is very comprehensive in each step. It even includes a preview of the final application. Unfortunately, it fails in small details. The user has to edit several configuration files by hand. These files' content sometimes differs from what is described in the text. Therefore, one has to make some guesses of how to modify the configuration.

An external UML editor is used for modeling the application. The author choose ArgoUML [Col], but several other tools are supported too. This is a very good approach. Users can use there favorite UML editor and are not limited by some internal policy of AndroMDA.

Unfortunately, the tutorial's content strongly differs from the behavior of ArgoUML. It was not possible to create a new project in the editor as described by the text.

Conclusion. AndroMDA would have been an interesting try, because it closely follows the idea of using common UML tools for modeling. Unfortunately, AndroMDA's tutorial did not work. After having to start twice and editing obscure configuration files, the author finally gave up when he encountered a problem with creating a new UML project, which did not work as advertised.

4.3 openArchitectureWare

openArchitectureWare [opeb] is another framework that provides MDA modeler and generator facilities. The website highlights several features including

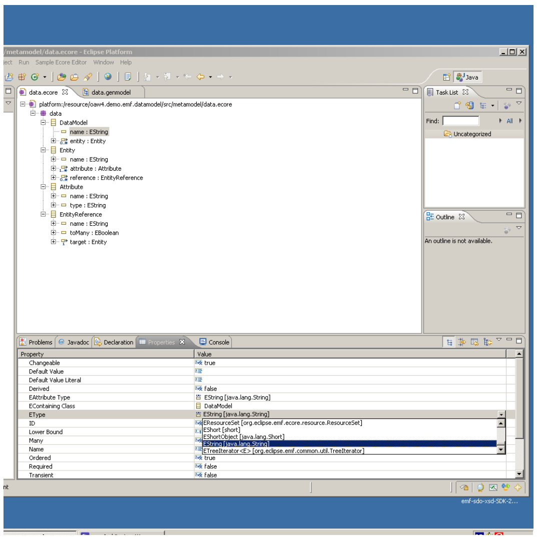
- transformations between models and text,
- Modeling front end,
- full EMF integration,
- full support for UML 2
- syntax checking, and
- an online help system.

openArchitectureWare is free software under the terms of the *Lesser General Public License.*

Installation. For this test, version 4.3 of the openArchitectureWare SDK is used. The package consists of an archive file which has to be extracted into Eclipse's directory. It is then automatically integrated into Eclipse.

Basic usage. openArchitectureWare is build on top of Eclipse's native EMF facilities. The tutorial [opea] describes the creation of a meta model that is successively transformed into running code.

The tutorial's first step is to build a meta model using Ecore, EMF's built-in meta-modeling package. For this task, Eclipse contains a graphical editor, which is shown in figure 3. Following the tutorial, this process is very easy, but feels a bit cumbersome.



```
platform:/resource/oaw4.demo.emf.datamodel/src/metamodel/data.ecore
  data
    DataModel
      name : EString
      entity : Entity
    Entity
      name : EString
      attribute : Attribute
      reference : EntityReference
    Attribute
      name : EString
      type : EString
    EntityReference
      name : EString
      toMany : EBoolean
      target : Entity
```

An outline is not available.

Property	Value
Changeable	true
Default Value	
Default Value Literal	
Derived	false
EAttribute Type	EString [java.lang.String]
EContaining Class	DataModel
EType	EString [java.lang.String]
ID	
Lower Bound	
Many	
Name	
Ordered	true
Required	false
Transient	false

```
EResourceSet [org.eclipse.emf.ecore.resource.ResourceSet]
EShort [short]
EShortObject [java.lang.Short]
EString [java.lang.String]
ETreeIterator<E> [org.eclipse.emf.common.util.TreeIterator]
```

Figure 3: The Ecore editor. The upper area contains elements of the meta model, the lower area contains their properties.

With the meta model in place, it is possible to generate the modeling code. This is done automatically by EMF. The result is a plugin which is then executed in Eclipse. Running the plugin allows to model a concrete problem. In the case of the tutorial, the problem consists of vehicles and persons, where each person should be associated with a vehicle.

Modeling the problem is again done in Eclipse. openArchitectureWare provides its own project type. An instance thereof is associated with the modeling code that was built before. Again, a graphical editor turns up that allows for the setup of different entities and their attributes.

Having modeled the problem, it should now be possible to execute openArchitectureWare's code generator and create an application. Unfortunately, Eclipse was not able to find the meta-model plugin. It was built and loaded, but could not be included into the dependency settings of the model plugin. This step is shown in the tutorial, but was not actually possible. The author ran the code generator without the correct dependency settings, but it failed with an error message.

Conclusion. The installation is very easy. Extracting Eclipse and openArchitectureWare into the same directory is all that needs to be done.

Creating the meta model and the model in the editor is also very easy, but it was found to be a rather repetitive task. In contrast to the Eclipse test, extending this model should not have been hard. The graphical model editor would have been the place of choice.

Unfortunately, it was not possible to complete the test. It is not clear whether the problem with the dependency settings was related to openArchitectureWare or Eclipse. Probably, an experienced user of Eclipse would have been able to resolve this problem.

openArchitectureWare looks quite mature. The author's conclusion is that readers, who have some experience with Eclipse and want to try out MDA, should take a look at openArchitectureWare.

5 Critique

Model Driven Architecture has received some serious critique. This section presents the critic's main points. Some articles can be found online at [Hay], [Tho], and [Fow].

UML. The use of UML is not suitable for complete modeling of applications because 1. UML does not fully define semantics and 2. UML is not sufficient for every problem.

The OMG offers two solutions to the problem of missing semantics. First, it is possible to define semantics with *Precise Action Semantics*, i.e. language constructs for specifying semantics of UML models. These languages differ between products from different vendors. While the model is independent from the platform it now depends on the transformation tool. The second solution is to leave semantics out of the PIM and manually add code to the generated PSM, which then implements semantics. This has do be done whenever the PSM changes. Both solutions strongly contradict the ideas of model-driven software development.

UML is insufficient for general use because it is targeted towards object-oriented designs. For example, modeling a parser generator, as described in the introduction, is not possible with UML. This insufficiency can be handled by creating new domain-specific languages. Defining a DSL needs the modeling tools to fully support MOF, which is often not the case.

Tools. The tool support for MDA is insufficient. Many tools do not fully support the Meta Object Facility. The basic standards UML and XMI are widely supported, but full MOF support requires the possibility to create new DSLs. Combined with the problems of UML this makes MDA lose a lot of expressiveness.

Usefulness. Another critique is related to the general usefulness of MDA. One major motivation behind MDA is the separation of an application's business logic from the underlying platform.

The critic's point is that this only makes sense if the platform ages much faster then the application. This seldom happens. Successful platforms are developed and maintained for decades while applications easily change within years. The fact that many platforms have open-source implementations[4] makes it even more likely that an application's platform will outlive the application itself.

Initial cost. The initial costs for MDA-based development are quite high. Instead of only creating an application's logic, i.e. the PIM, the developers also need to define the necessary transformations from PIM to PSM. The fact that UML is not sufficient for every scenario or unable to fully provide semantics makes the situation even worse.

This extra development effort is likely to increase the *time to market*, i.e. the time between the planning of a product and its launch. The longer the product is in development, the more investment it takes, and the more likely it is that competitors release similar products. At the end, the profit margin might be smaller if an MDA-based development approach was chosen.

6 Conclusion

An overview of Model Driven Architecture was presented. The paper started with some widely-used examples of modeling and model-driven development in Section 1. It presented some MDA related technologies and predecessors in Section 2. In Section 3, MDA's concepts of modeling and transformation were discussed. The author's

[4] Java and .Net are the obvious examples here.

personal experiences with some MDA tools were described in Section 4. Section 5 summarized the critique of MDA-based development.

References

[Anda] AndroMDA: *AndroMDA.org - Getting started Java.* `http://galaxy.andromda.org/index.php?option=com_content&task=category&sectionid=11&id=42&Itemid=89`, visited on 27th May 2008.

[Andb] AndroMDA: *AndroMDA.org - Home.* `http://www.andromda.org`, visited on 27th May 2008.

[ASUL06] Aho, Alfred V., Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam: *Compilers: Principles, Techniques, and Tools.* Pearson Education Inc., 2nd edition, 2006.

[Col] CollabNet: *argouml.tigris.org.* `http://argouml.tigris.org`, visited on 27th May 2008.

[Ecla] Eclipse: *Eclipse.org home.* `http://www.eclipse.org`, visited on 27th May 2008.

[Eclb] Eclipse: *Help - Eclipse SDK.* `http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.emf.doc/tutorials/clibmod/clibmod.html`, visited on 27th May 2008.

[Fow] Fowler, Martin: *ModelDrivenArchitecture.* `http://martinfowler.com/bliki/ModelDrivenArchitecture.html`.

[GJSB05] Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha: *The Java Language Specification.* Prentice Hall, 3rd edition, 2005.

[GLA] GLADE: *Glade - a User Interface Designer for GTK+ and Gnome.* `http://glade.gnome.org/`, visited on 11th May 2008.

[GNU] GNU: *Bison - GNU parser generator.* `http://www.gnu.org/software/bison/`, visited on 3rd May 2008.

[Hay] Haywood, Dan: *MDA: Nice idea, shame about the...* `http://www.theserverside.com/tt/articles/article.tss?l=MDA_Haywood`.

[Int96] International Organization for Standardization: *ISO/IEC 14977.* International Organization for Standardization, 1996. `http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`, visited on 3rd May 2008, Extended Backus-Naur form.

[Int05] International Organization for Standardization: *ISO/IEC 19501.* International Organization for Standardization, 2005. `http://www.omg.org/spec/UML/ISO/19501/PDF/`, visited on 27th May 2008, UML version 1.4.2.

[Khr] Khronos: *OpenGL.* `http://www.opengl.org`, visited on 25th May 2008.

[OMGa] OMG: *MDA*. `http://www.omg.org/mda`, visited on 3rd May 2008.

[OMGb] OMG: *Object Management Group*. `http://www.omg.org`, visited on 3rd May 2008.

[OMGc] OMG: *Object Management Group - UML*. `http://www.uml.org`, visited on 3rd May 2008.

[OMGd] OMG: *Welcome To The OMG's CORBA Website*. `http://www.omg.org/corba`, visited on 3rd May 2008.

[opea] openArchitectureWare: *oAW Tutorial*. `http://www.eclipse.org/gmt/oaw/doc/4.3/html/contents/emf\_tutorial.html`txurldatecomment 27th May 2008.

[opeb] openArchitectureWare: *openArchitectureWare.org - Official openArchitectureWare Homepage*. `http://www.openarchitectureware.org`, visited on 27th May 2008.

[Str00] Stroustrup, Bjarne: *The C++ Programming Language*. Addison-Wesley Professional, 3rd edition, 2000.

[Tho] Thomas, Dave: *UML - Unified or Universal Modeling Language?* `http://www.jot.fm/issues/issue_2003_01/column1/`, visited on 27th May 2008.

[W3C] W3C: *Extensible Markup Language (XML)*. `http://www.w3.org/xml`, visited on 27th May 2008.